METHOD AND APPARATUS FOR EMULATING SIMD INSTRUCTIONS

I.   FIELD

The present invention relates to digital computer systems, and more particularly but not by way of limitation, to methods and an apparatus for processing instructions in such systems.

II.   BACKGROUND

Microprocessors exist that implement a reduced instruction set computing (RISC) instruction set architecture (ISA) and an independent complex instruction set computing (CISC) ISA by emulating the CISC instructions with instructions native to the RISC instruction set. Instructions from the CISC ISA are called "macroinstructions." Instructions from the RISC ISA are called "microinstructions."

Streaming Single-Instruction Multiple-Data Extensions (SSEs) have been developed to enhance the instruction set of the latest generation of certain computer architectures, for example the IA-32 architecture. The SSEs include a new set of registers, new floating point data types, and new instructions. Specifically, the SSEs comprise eight 128-bit Single-Instruction Multiple-Data (SIMD) floating point registers (XMM0 through XMM7) that can be used to perform calculations and operations on floating point data. These XMM registers are shown in Figure 1A. Each 128-bit floating point register contains four packed 32-bit single precision (SP) floating point (FP) numbers. The structure of the packed 32-bit SP FP numbers is illustrated in the example of Figure 1B, where four 32-bit SP FP numbers (numbered 0 through 3) are shown as if stored in the XMM2 SSE register. In architectures designed to support the SSEs, such as its native architecture, a single instruction of the SSE instruction set operates in parallel on the four 32-bit SP FP numbers in a particular XMM register.

The SSEs also include a status and control register called the MXCSR register. The format of the MXCSR is illustrated in the example of Figure 1C. The MXCSR register may be used to selectively mask or unmask exceptions. Specifically, bits 7-12 of the MXCSR register may be used by a programmer to selectively mask or unmask a particular exception. Masked exceptions are those exceptions that a programmer wishes to be handled automatically by the processor which may provide a default response. Unmasked exceptions, on the other hand, are

1   those exceptions that the programmer wishes to be handled by invocation of an
2   interrupt or an operating system handler. This invocation of the handler transfers
3   control of the operating system (such as Windows by Microsoft) where the problem
4   may be corrected or the program terminated.

5   The MXCSR register may also be used to keep track of the status of exception
6   flags. Bits 0-5 of the MXCSR register indicate whether any of six exceptions have
7   occurred in the execution of an SSE instruction. Those exceptions include the
8   following: invalid operation (I), divide-by-zero (Z), denormal operation (D), numeric
9   overflow (O), numeric underflow (U), or inexact result (P). The status of the flags
10   are "sticky," meaning that once they are set, they are not cleared by any subsequent
11   SSE instruction, even if one is performed without exception. The status flags can
12   only be cleared by a special instruction, usually issued from the operating system.

13   The exception flags of Figure 1C are the result of a bitwise logical-OR
14   operation on all four of the 32-bit SP FP operations that are performed on a particular
15   128-bit register XMM register. (One operation on each of the four 32-bit SP FP
16   numbers.) Thus, if an exception occurs as to any one of the four 32-bit SP FP
17   numbers, the exception flag for that particular type of exception will be raised,
18   indicating some type of problem has occurred in the system. The invalid operation
19   (I) divide by zero (Z), and denormal operation (D) exceptions are pre-computation
20   exceptions, meaning that they are detected before any arithmetic or logical operations
21   occur. That is, they can be detected without doing any computations. The other three
22   exceptions, numeric overflow (O), numeric underflow (U), and inexact result (P) are
23   post-computation exceptions, meaning that they are detected after the operations have
24   been performed. It is possible for an operation performed on a sub-operand (i.e., one
25   of the four operands in a 128-bit XMM register) to raise multiple flags.

26   SSEs have the following rules for exceptions:

27   1.   When an unmasked exception occurs, the processor executing the
28   instruction will not change the contents of the XMM register. In other words, results
29   will not be committed or stored until it is known that no unmasked exceptions have
30   occurred with respect to any of the four 32-bit SP FP numbers.

31   2.   If there is a masked exception, all exception flags are updated.

1   3.   In the case of unmasked pre-computation exceptions, all flags relating

2   to pre-computation exceptions, whether masked or unmasked, will be updated.

3   However, no subsequent computations are permitted, meaning that no post-execution

4   exceptions can or will occur. This, of course, means that no post-execution exception

5   flags will change or be updated.

6   4.   In the case of unmasked post-computation exceptions, all post-

7   execution conditions, whether masked or unmasked, will be updated, as will all pre-

8   computation exceptions. Any pre-computation exceptions will be masked exceptions

9   only because, if the pre-computation exception was unmasked, under Rule No. 3

10   above, no further computations would have been permitted.

11   Further information regarding streaming SIMD extensions may be found in

12   the Intel Architecture Software Developer's Manual, (1999), Volumes 1 through 3,

13   Intel Order Numbers 243190, 243191, 243192, which are hereby incorporated by

14   reference.

15   In many architectures, provisions have not been made for the SSE

16   instructions. In these non-native architectures, the eight 128-bit floating point XMM

17   registers capable of containing four 32-bit SP FP numbers are not available. In some

18   non-native architectures, the eight 128-bit XMM registers may be mapped onto 16

19   floating point registers (e.g., IA-64 registers) that may be less than 128 bits and more

20   than 64 bits wide. Specifically, some architectures use 82-bit registers to hold two

21   32-bit SP FP numbers (the bits in excess of 64 may be used for the special encoding

22   used to indicate the register holes SIMD-type 32-bit SP FP numbers). An example

23   is shown in Figure 1D. Note that the four 32-bit SP FP numbers 0-3 stored in the

24   XMM2 register of the SSE native environment (Figure 1B) are now stored in two 82-

25   bit registers, XMM2_Low and XMM2_High, containing the "low half of the XMM2

26   register" and the "high half of the XMM2 register," respectively. This makes parallel

27   execution of an operation on each of the four 32-bit SP FP numbers difficult.

28   Thus, in this non-native environment, the SSE instructions must be executed

29   by emulation. Specifically, operations may first be performed on two of the four 32-

30   bit SP FP numbers (in parallel) and then be performed on the remaining two 32-bit

31   SP FP numbers (again, in parallel). Operations may alternatively be performed on

32   only one or at least three of the 32-bit SP FP numbers. For example, an operation

1   may be performed on the operands in the low half, XMM2_Low, and then on the high

2   half, XMM2_High. However, given the SSE rules for handling exceptions and

3   updating exception flags, problems arise when emulating SSE instructions in this

4   partially parallel, partially sequential manner. For example, consider a set of

5   instructions being performed on the low half and high half of Figure 1D:

6   XMM2 := OP(XMM3, XMM4)

7   emulated by

8   XMM2_Low := OP(XMM3_Low, XMM4_Low)

9   XMM2_High := OP(XMM3_High, XMM4_High)

10   Assume that the first instruction is executed without an unmasked exception as to the

11   operands in the low halves, XMM3_Low and XMM4_Low. The results of this

12   operation are then properly committed in XMM2_Low. Assume now that execution

13   of the second instruction on the high halves results in a pre-computation unmasked

14   exception. According to the SSE rules, no subsequent operations are then to be

15   performed on any of the four 32-bit SP FP numbers because of that pre-computation

16   unmasked exception. But here, however, results of the operation on the low halves

17   have been committed to register XMM2_Low in violation of the SSE rules. This

18   corrupts the data in XMM2_Low and cannot be allowed to happen.

19   Prior machines have solved this problem by implementing a "back-off"

20   mechanism that allowed them to speculatively change architectural state when the

21   first microinstruction completed, then "undo" the change if the second

22   microinstruction had an exception. This back-off mechanism may be hard to

23   implement in certain machines, especially in machines that do not implement register

24   renaming. In many systems, the use of a back-off or undo operation is difficult or

25   limited, for various reasons.

26   One way of successfully emulating the SSEs and preventing this rule violation

27   is to use a "shadow" register mechanism. In a shadow register mechanism, the results

28   of a previous, successful operation on the low halves are physically stored in a

29   shadow register. In this case, in the example above, when the exception is detected

30   on the high halves, the results previously stored in the shadow register for the

31   previous operation on the low halves may be re-stored, that is, an "undo" operation

32   on the low halves as performed. The shadow register mechanism, however, is

1 relatively complex. In most systems, there must be at least 16 registers available for
2 storing the results of a previous operation on the low halves, and each must be
3 capable of two 32-bit FP SP numbers. Additionally, when an "undo" operation is
4 required, it must be determined which of the shadow registers the desired results are
5 in. This mechanism consumes valuable register space that could otherwise be used
6 more efficiently. Furthermore, a relatively complicated system of pointers and virtual
7 maps are required to store the previous maps.

8 Another way to emulate a particular SSE instruction is to provide a back-off
9 register mechanism. One skilled in the art will recognize that this technique may
10 require a plurality of registers, a multiplexor and a de-multiplexor combination,
11 various other hardware, and a new set of instructions. All of these increase cost and
12 reduce efficiency.

13 Yet another way to emulate a particular SSE instruction is to execute the
14 instruction with respect to each of the four 32-bit SP FP numbers in the SSE XMM
15 register one at a time and store the results of each execution in temporary registers.
16 When the instruction has been executed with respect to the fourth 32-bit SP FP
17 number, and no unmasked exceptions have occurred, the results may then be
18 committed to the appropriate architectural location and exception flags be updated.
19 This method of emulation requires the addition of a relatively complex micro-code
20 sequence and the use of hardware that could otherwise be used more efficiently, not
21 to mention the amount of clock cycles it consumes in executing an instruction four
22 times before results can be committed.

23 Clearly, there exists a need for a method and an apparatus for emulating the
24 SSE instruction set (and other instruction sets) that makes efficient use of existing
25 hardware and that consumes relatively few clock cycles. Additionally, there exists
26 a need for a method and apparatus for determining whether certain problems may
27 occur in the execution of a series of instructions without committing the results of
28 those instructions.

29 **III. <u>SUMMARY</u>**

30 The present invention is a method for processing instructions by decomposing
31 a macroinstruction into at least two microinstructions, executing the microinstructions
32 in parallel on two separate functional units, and linking the microinstructions such

1 that they appear as though they were executed as a single functional unit. The present

2 invention operates by determining whether certain exceptions occur in either of the

3 functional units, according to SSE rules for exceptions. If an exception does occur

4 in any of the linked microinstructions, then the execution of each of those

5 microinstructions is canceled. This avoids the necessity of a backoff or undo

6 mechanism.

7 The present invention is also a computer system for processing software

8 instructions, having a processor with a floating point unit, a ROM, and floating point

9 registers. The processor is configured to decompose a macroinstruction into at least

10 two microinstruction, to execute those microinstructions in parallel, and to link those

11 instructions such that they appear to execute as a single functional unit. The

12 processor also is capable of identifying and treating exceptions without the use of a

13 back-off or undo mechanism.

IV. BRIEF DESCRIPTIONS OF THE DRAWINGS

15 Figures 1A through 1D are block diagrams of components of the SSEs.

16 Figure 2 is a block diagram of a computer system including the present

17 invention.

18 Figure 3 is a block diagram of the processor in Figure 2.

19 Figure 4 is a flow chart of the method of the present invention.

V. DETAILED DESCRIPTION

A. The Computer System

22 Figure 2 illustrates a computer system 10 in which the present invention may

23 be implemented. The computer system 10 comprises at least one processor 20, main

24 memory 30, and various interconnecting data, address, and control busses (numbered

25 collectively as 40). An instruction set 50, which may include SSEs, and an operating

26 system 60 may be stored in main memory 30.

27 As illustrated in Figure 3, the processor 20 comprises a floating point unit 70,

28 a micro-code ROM 100, various busses and interconnections (numbered collectively

29 as 110) and a register file 120 comprising the sixteen floating point registers,

30 XMM0_Low through XMM7_High, needed to emulate the SSE XMM registers. In

31 one embodiment, the sixteen floating point registers are 82-bit registers, but other

32 widths (e.g., 128-bit or 64-bit) may be used, as the following description in terms of

1   82-bit registers is exemplary only, and not intended in a limiting sense. The four 32-

2   bit SP floating point numbers of the SSEs may be stored in two of the 82-bit SP FP

3   registers of the present invention (e.g., XMM2_Low and XMM2_High, as illustrated

4   in Figure 1D). The floating point unit 70 comprises a first 32-bit register 130 that

5   corresponds to the MXCSR register of the SSEs and at least two functional units (e.g.,

6   170, 172) used to execute two FP operations.

7   Instructions are provided to the processor 20 from main memory 30. The

8   instructions provided to the processor 20 are macro-code instructions that map to one

9   or more micro-code instructions 140 stored in the micro-code ROM 100. The micro-

10   code instructions can be directly executed by the processor 20. Also stored in the

11   micro-code ROM 100 are a set of micro-code handlers 150 that may be invoked to

12   handle certain unmasked processor exceptions. The processor 20 may have a

13   pipelined architecture and may allow for parallel processing of certain instructions.

14   **B.   In Use**

15   The IA-32 ISA defines streaming SIMD extensions, or SSE instructions, that

16   allow a single instruction to operate simultaneously on multiple single-precision (SP)

17   floating-point (FP) values held in a register file that is eight entries deep and 128 bits

18   wide. Since single-precision floating-point numbers can be represented in 32 bits,

19   each SSE register can hold four packed SP values, and each SSE instruction can

20   calculate four SP results. Even though the IA-32 ISA specifies 128-bit registers and

21   instructions that can operate on 128 bits of data, an implementation may choose to

22   implement narrower registers or FP functional units that can only calculate one or two

23   FP SP operations at a time. It is possible, then, to implement the SSE instructions by

24   emulating them on multiple functional units by executing the four FP SP operations

25   serially over multiple cycles, or by some combination of both of these techniques.

26   The ISA only requires that any architecturally visible side effects of the operations

27   appear as though all four operations occurred concurrently in the hardware.

28   In a preferred embodiment, a RISC ISA is implemented containing FP

29   registers that are 82 bits wide. Thus, to emulate SSE instructions on this

30   implementation, two FP registers are required to emulate the requisite four 32-bit SP

31   FP values contained in one SSE register. Also, the RISC ISA defines its own SIMD

32   operations that operate on two SP values with one instruction. So in this

1   implementation, two RISC microinstructions are required to emulate a single SSE

2   macroinstruction.

3   SSE instructions may cause exceptions when they execute, depending on the

4   values of the operands. The IA-32 architecture defines six possible exceptions, three

5   pre-computation and three post-computation. Pre-computation exceptions are those

6   which are calculated based on the values of the source registers, whereas post-

7   computation exceptions are calculated based on the value of the result of the

8   computation. The architecture also defines a control and status register, the MXCSR.

9   The MXCSR contains control bits that can independently mask each exception and

10   flag bits that capture the status of any exceptions that occur.

11   A preferred embodiment of the present invention includes a method for

12   issuing two RISC microinstructions in parallel, a method for updating the flags based

13   on the results of both FP units, and a method for causing an exception in either FP

14   unit to inhibit setting a result register in both FP units.

15   When executing in native mode (RISC ISA), the implementation is required

16   to have precise exceptions. That is, if an instruction causes an exception, then all

17   subsequent instructions in the instruction stream must be flushed from the process.

18   Even if an implementation is executing multiple instructions in parallel, the sequential

19   semantics must be adhered to. For example, in the following instruction stream:

20   op1 - oldest instruction

21   op2

22   op3

23   op4

24   . . .

25   opN - youngest instruction

26   ~~Op1 is the oldest instruction, and if it takes a fault, the processor must flush~~

27   all younger operations (op2 - opN). If the processor executes two operations in

28   parallel and pipelines new operations every cycle, all these operations might be "in

29   flight" at the same time. For example, op1 and op2 might have issued in parallel and

30   may be close to completion, while op3 and op4 are in flight one cycle behind, op5 and

31   ~~op6 are in flight two cycles behind, etc. If op1 takes an exception, all operations (op1~~

Figure 4 is a flow chart showing the steps required to implement the present invention, to emulate SSE instructions. First, a macroinstruction is decomposed into two or more microinstructions in a decompose function 310. The two microinstructions used to emulate the high half and the low half of the SSE operation are forced to issue in parallel and are dispatched simultaneously to the two FP units each capable of calculating two SP values per cycle, as shown in the emulation function 320. This is accomplished via a mechanism claimed in copending patent application, entitled "Method and Apparatus for Implementing Two Architectures on a Chip," inventors Knebel, et al., filed on the same date as this application, which is hereby incorporated by reference. Both micro-operations must be treated as a single operation. They move in lockstep with each other.

A signal is then generated by the emulation hardware and sent to the FP functional unit to specify when an SSE instruction is being emulated, as shown in the signal generation function 330. Then, when a signal fires, an exception detection function 340 determines whether an exception is taken in either FP functional unit. If an exception is taken in either FP functional unit, then the results from both FP functional units must be cleared, the execution is canceled, and a handler is invoked, as shown by the cancellation function 350. The invention must be capable of flushing the result in the other FP functional units, regardless of the relative age of the microinstructions in the two functional units. When a signal above fires, the MXCSR flag must be updated based on the results of both functional units, as shown in the update function 360. Following the successful completion of the execution of the microinstructions, the process continues, as shown in the continuation function 370.

Although the present invention has been described in detail with reference to certain embodiments thereof, variations are possible. Therefore, the present invention may be embodied in other specific forms without departing from the essential spirit or attributes thereof. It is desired that the embodiments described herein be considered in all respects as illustrative, not restrictive, and that reference be made to the appended claims for determining the scope of the invention.